Engineering of Advanced Software Solutions (EASS)

HIT, Israel

Yossi Eliaz

2022

EASS 2022 Part B

- Admin

1. Introduction to the course, ways to make an online project Medium, linkedin, heroku How to write a readable and informative post
2. Familiarity with latex and project documentation How to write articles in latex, how to write in mathematical language in google colab or in MARKDOWN Overleaf, Markdown 3-5. methods for storing data in large systems (feature analyzes, differences, etc.) Introduction to types of database systems: SQL, NOSQL, KEY-VALUE, GRAPH DATABASE STATEFULL systems REDIS, monogodb, mysql sqlalchemy, pydantic, ORM 6-7. methods for processing data in bulk Working with tabular information, and introduction to Xgoost trees, pandas 8-9. Methods for Integrating Advanced Algorithmics in Pytroch CNN Convolution Network Systems
3. Advanced Testing How to write UNIT TESTS and integration tests Demonstration in Python and BASH
4. Advanced debugging How to debug system, software in multiple processes Methods for debugging in different environments
5. Advanced Profiling What is profiling and how is it profiled in the Linux and Python environment
6. Technological analysis and the connection between business logic and business model in the world of software engineering Business models for software products, the connection between them and the software architecture of large companies (Netflix, Uber, etc.) Overview of the various groups

EASS 2022 Part B

- Familiarity with modern software development and software methodologies
- Meta-programming (ancillary software development tools other than the software code itself)
- Debugging and profiling of large systems
- Work with tabular as well as NOSQL databases
- Introduction to graphical databases
- Introduction to advanced frontend and backend systems
- Systems design with advanced algorrithmic logic, such as:
- Xgboost based solutions
- Convolution Networks
- Use of fastapi
- References to business models for software ventures
- Presentation of software projects

EASS 2022 Part B

- Bitcoin mechanics
- Digital signatures
- Bitcoin scripts
- Wallets
- Consensus protocols (Nakamoto consensus, large scale consensus, VDFs)
- Ethereum mechanics
- Solidity
- MultiChain
- Fintech
- Stablecoins and oracles
- Decentralized exchanges
- DeFi lending systems
- Privacy
- zk-SNARKs
- Scaling the blockchain (payment channels, state channels, rollup)
- Recursive SNARKs
- rust, low level, gdb, kernel modules,llvm, gcc, webassembly, CUDA

EASS
HIT

Admin stuff

::::::::::::: {.columns} ::: {.column width="50%"} - Discord - Github account - HW on Github - Creating a Canvas account (https://canvas.instructure.com/) - Accepted invitation from AWS Academy and GitHub (I have sent links) - AWS on Cavnvas - LinkedIn - Commandline (WSL) - Docker - k8s/github actions - Moodle (minimal interaction over there) - Stackoverflow - Engagment on Discord - Hackernews - Getting our desired job/start a company

ReactJS Web-UI

Installing

```
brew install node
npx create-react-app my-app
cd my-app
npm start
```

ReactJS example `App.js` (1/2)

```
import React, { Component } from "react";
import logo from "./logo.svg";
import "./App.css";

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      inputText: "",
      counter: 0,
    };
  }

  handleClick = (event) => {
    console.log(event.target);
    const val = event.target.name === "Up" ? 1 : -1;
    this.setState({
      counter: this.state.counter + val,
    });
  };

  handleChange = (event) => {
    this.setState({
      inputText: event.target.value,
    });
```

ReactJS example App.js (2/2)

```
handleGet = (event) => {
  fetch("http://httpbin.org/get")
    .then((response) => response.json())
    .then((data) => console.log(data));
};
```

ReactJS example App.js (3/3)

```
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} alt="Logo" width="10%" />
          <input type="text" onChange={this.handleChange} value={this.state.inpu
          <p>{this.state.inputText}</p>
          <button name="Up" onClick={this.handleClick}>
            Up
          </button>
          <p>{this.state.counter}</p>
          <button name="Down" onClick={this.handleClick}>
            Down
          </button>
          <p>Perform HTTP GET to httpbin</p>
          <button onClick={this.handleGet}>HTTP get</button>
        </header>
      </div>
    );
  }
}

export default App;
```

```
# Testing
```

More examples

```python
import asyncio
from pyppeteer import launch


async def main():
    browser = await launch()
    page = await browser.newPage()

    await page.goto('https://example.com')

    # J is an alias to querySelector
    input_el = await page.J('input[type="text"]')
    await input_el.type("Puppeteer")

    submit_el = await page.J('input[type="submit"]')
    await submit_el.click()

    assert 'Puppeteer' in (await page.content())

    browser.close()


asyncio.get_event_loop().run_until_complete(main())
```

Security concepts

1. Password strength:

The strength of a password is determined by its length, complexity, and uniqueness. To ensure that passwords are strong, web applications should require passwords to be a minimum of eight characters in length and include a mix of uppercase and lowercase letters, numbers, and special characters. Passwords should also not be common words or easily guessed phrases.

2. Encryption:

All data transmitted between the web server and web browser should be encrypted to protect it from being intercepted and read by third parties. Transport Layer Security (TLS) is the most common protocol used for encryption. When TLS is used, a padlock icon will typically appear in the web browser to indicate that the connection is secure.

TLS(Transport Layer Security) is a cryptographic protocol that provides communication security over the Internet. It has two main components: a public-key Infrastructure (PKI) to verify the identity of endpoints, and a symmetric-key mechanism to encrypt/decrypt data.

Intro to encryption

There are two types of encryption schemes: private-public key encryption and shared key encryption.

Private-public key encryption, also known as asymmetric key encryption, is a type of encryption where there are two different keys - a public key and a private key. The public key can be known by anyone and is used to encrypt data. The private key is only known by the recipient and is used to decrypt data.

Shared key encryption, also known as symmetric key encryption, is a type of encryption where there is only one key that is shared between the sender and the recipient. This key is used to both encrypt and decrypt data.

Symmetric encryption

```
pip install cryptography
```

```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
key = os.urandom(32)
iv = os.urandom(16)
cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
encryptor = cipher.encryptor()
ct = encryptor.update(b"a secret message") + encryptor.finalize()
decryptor = cipher.decryptor()
decryptor.update(ct) + decryptor.finalize()
```

Hash-based message authentication codes (HMAC)

```
from cryptography.hazmat.primitives import hashes, hmac
key = b'test key. Beware! A real key should use os.urandom or TRNG to generate'
h = hmac.HMAC(key, hashes.SHA256())
h.update(b"message to hash")
signature = h.finalize()
print(signature)
```

Verify

```
h = hmac.HMAC(key, hashes.SHA256())
h.update(b"message to hash")
h_copy = h.copy() # get a copy of `h' to be reused
h.verify(signature)
```

Private-Public key (RSA)

```
from cryptography.hazmat.primitives.asymmetric import rsa
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
)
public_key = private_key.public_key()
```

Private-Public key (RSA) - signing

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
message = b"A message I want to sign"
signature = private_key.sign(
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
```

Private-Public key (RSA) Sign-Verify

Sign

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
message = b"A message I want to sign"
signature = private_key.sign(
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
```

Verify

```
public_key = private_key.public_key()
public_key.verify(
    signature,
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
```

Private-Public key (RSA) Encrypt-Decrypt

Encrypt

```
message = b"encrypted data"
ciphertext = public_key.encrypt(
    message,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
```

Decrypt

```
plaintext = private_key.decrypt(
    ciphertext,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
plaintext == message
```

TLS example in python

TLS(Transport Layer Security) is a cryptographic protocol that provides communication security over the Internet. It has two main components: a public-key Infrastructure (PKI) to verify the identity of endpoints, and a symmetric-key mechanism to encrypt/decrypt data.

```
openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout key.pem
```

client.py

```python
from socket import create_connection
from ssl import SSLContext, PROTOCOL_TLS_CLIENT


hostname = "HIT.COM"
ip = "127.0.0.1"
port = 8443
context = SSLContext(PROTOCOL_TLS_CLIENT)
context.load_verify_locations("cert.pem")

with create_connection((ip, port)) as client:
    with context.wrap_socket(client, server_hostname=hostname) as tls:
        print(f"Using {tls.version()}\n")
        tls.sendall(b"PING")

        data = tls.recv(1024)
        print(f"Server says: {data}")
```

server.py

```python
from socket import socket, AF_INET, SOCK_STREAM
from ssl import SSLContext, PROTOCOL_TLS_SERVER


ip = "127.0.0.1"
port = 8443
context = SSLContext(PROTOCOL_TLS_SERVER)
context.load_cert_chain("cert.pem", "key.pem")


with socket(AF_INET, SOCK_STREAM) as server:
    server.bind((ip, port))
    server.listen(1)

    with context.wrap_socket(server, server_side=True) as tls:
        while True:
            connection, address = tls.accept()
            print(f"Connected by {address}\n")

            data = connection.recv(1024)
            print(f"Client Says: {data}")

            connection.sendall(b"PONG")
```

login with google OAuth

```
$ npx create-react-app demo-login-google
$ npm install --force react-google-login # to make sure react version is compati
$ cd demo-login-google
$ npm start
```

```
import logo from './logo.svg';
import './App.css';

import React from 'react';
import GoogleLogin from 'react-google-login';

function Login() {

  const responseGoogle = (response) => {
    console.log(response);
  };

  return (
    <div>
      <GoogleLogin
        clientId="973485012429-jdemkanneidu38bp284i1rr0hoi9uojr.apps.googleuserc
        buttonText="Login"
        onSuccess={responseGoogle}
        onFailure={responseGoogle}
        cookiePolicy={'single_host_origin'}
      />
```

login with google OAuth

```
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
      <Login />
    </div>
  );
}

export default App;
```

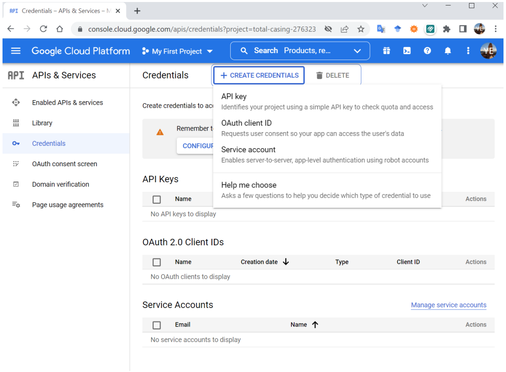
Using Google OAuth2

OAuth2 is an open standard for authorization that provides a way for users to grant third-party access to their resources without sharing their passwords. It works by delegating user authentication to the service that hosts the user's account, and authorizing third-party applications to access the user's resources using a standardized method.

1. Go to https://console.developers.google.com/projectcreate and create a project
2. Create credentials https://console.developers.google.com/apis/credentials
3. Choose OAuth client ID
4. Select Web application type

Using Google OAuth2

1. Go to https://console.developers.google.com/projectcreate and create a project
2. Create credentials https://console.developers.google.com/apis/credentials
3. Choose OAuth client ID
4. Select Web application type



Credentials

Create credentials to ac

Remember t

CONFIGUR

API key
Identifies your project using a simple API key to check quota and access

OAuth client ID
Requests user consent so your app can access the user's data

Service account
Enables server-to-server, app-level authentication using robot accounts

Help me choose
Asks a few questions to help you decide which type of credential to use

API Keys

Name	Actions
No API keys to display	

OAuth 2.0 Client IDs

Name	Creation date ↓	Type	Client ID	Actions
No OAuth clients to display				

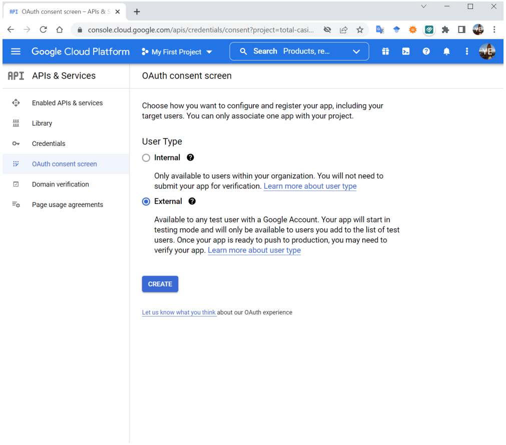
Service Accounts

Manage service accounts

Email	Name ↑	Actions
No service accounts to display		

Using Google OAuth2

1. Go to https://console.developers.google.com/projectcreate and create a project
2. Create credentials https://console.developers.google.com/apis/credentials
3. Choose OAuth client ID
4. Select Web application type



OAuth consent screen

Choose how you want to configure and register your app, including your target users. You can only associate one app with your project.

User Type

( ) Internal

Only available to users within your organization. You will not need to submit your app for verification. Learn more about user type

[✓ External]

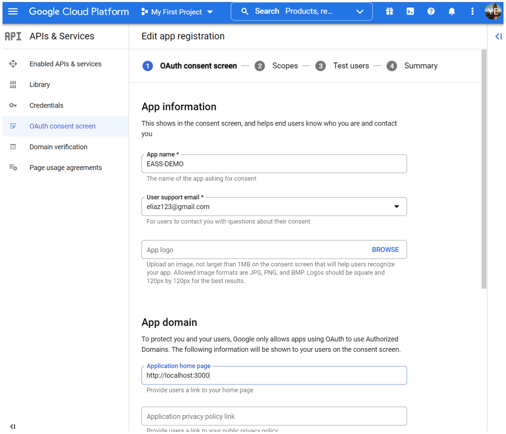
Available to any test user with a Google Account. Your app will start in testing mode and will only be available to users you add to the list of test users. Once your app is ready to push to production, you may need to verify your app. Learn more about user type

CREATE

Let us know what you think about our OAuth experience

Using Google OAuth2

1. Go to https://console.developers.google.com/projectcreate and create a project
2. Create credentials https://console.developers.google.com/apis/credentials
3. Choose OAuth client ID
4. Select Web application type



Edit app registration

1 OAuth consent screen — 2 Scopes — 3 Test users — 4 Summary

App information

This shows in the consent screen, and helps end users know who you are and contact you

App name *: EASS-DEMO
The name of the app asking for consent

User support email *: eliaz123@gmail.com
For users to contact you with questions about their consent

App logo BROWSE
Upload an image, not larger than 1MB on the consent screen that will help users recognize your app. Allowed image formats are JPG, PNG, and BMP. Logos should be square and 120px by 120px for the best results.

App domain

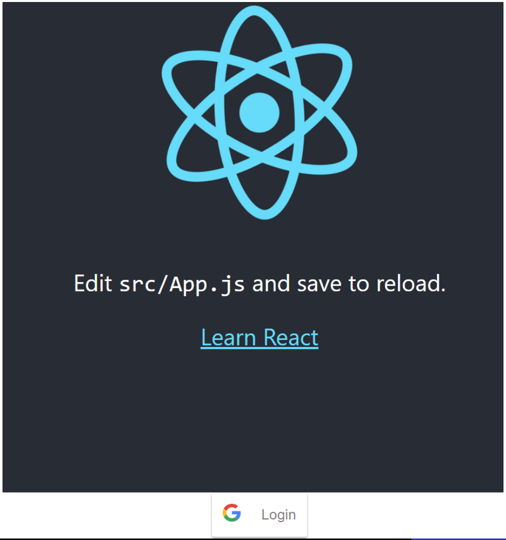
To protect you and your users, Google only allows apps using OAuth to use Authorized Domains. The following information will be shown to your users on the consent screen.

Application home page: http://localhost:3000
Provide users a link to your home page

Application privacy policy link
Provide users a link to your public privacy policy

Using Google OAuth2

1. Go to https://console.developers.google.com/projectcreate and create a project
2. Create credentials https://console.developers.google.com/apis/credentials
3. Choose OAuth client ID
4. Select Web application type



Edit `src/App.js` and save to reload.

Learn React

Security concepts

1. Authentication:

Users of a web application should be authenticated before being granted access to sensitive information or functionality. There are many different authentication methods that can be used, such as password-based authentication or two-factor authentication using a code generated by an app on a user's mobile device.

4. Authorization:

After a user has been authenticated, the web application needs to determine what they are authorized to do within the application. This usually involves assigning users to roles with different levels of access, such as Admin, Manager, or User. Once again, there are many different ways that this can be implemented depending on the requirements of the application.

Examples

There are many ways to secure a web application in node.js. Some common methods are discussed below.

1. Authentication and Authorization:

One of the most important aspects of security is authentication and authorization. Authentication is the process of verifying that a user is who they claim to be, while authorization is the process of verifying that a user has permission to access a particular resource.

There are many different ways to implement authentication and authorization, but one popular approach is to use JSON Web Tokens (JWTs). JWTs are an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

JWT is a standard to codify a JSON object in a long dense string without spaces. It is not encrypted, so, anyone could recover the information from the contents. But it's signed. So, when you receive a token that you emitted, you can verify that you actually emitted it.

`RS256` (RSA Signature with SHA-256) is an asymmetric algorithm, and it uses a public/private key pair: the identity provider has a private (secret) key used to generate the signature, and the consumer of the JWT gets a public key to validate the signature. Since the public key, as opposed to the private key, doesn't need to be kept secured, most identity providers make it easily available for consumers to obtain and use (usually through a metadata URL).

`HS256` (HMAC with SHA-256), on the other hand, involves a combination of a hashing function and one (secret) key that is shared between the two parties used to generate the hash that will serve as the signature. Since the same key is used both to generate the signature and to validate it, care must be taken to ensure that the key is not compromised.

Demo: https://jwt.io/ Video: https://www.youtube.com/watch?v=7Q17ubqLfaM Post: https://developer.okta.com/blog/2018/06/20/what-happens-if-your-jwt-is-stolen

```
//Create a new token with the payload //(usually this would be done after authen
const jwt = require('jsonwebtoken'); const token = jwt.sign({id: 1, username: 't
// eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwidXNlcm5hbWUiOiJ0ZXN0In0.-dnY
//Verify the token against the secret key
const jwt = require('jsonwebtoken'); const decoded = jwt.verify(token, 'secretke
//If the token is invalid, an error will be thrown
const jwt = require('jsonwebtoken'); try { const decoded = jwt.verify(token, 'se
```

Security concepts

2) Hashing and Salting:

Another important aspect of security is hashing and salting passwords. Hashing is the process of converting a password into a fixed length string of characters that cannot be reversed, while salting is the process of adding random data to the password before it is hashed in order to make it more difficult to crack by brute force methods.

There are many different algorithms that can be used for hashing and salting passwords, but one popular choice is bcrypt. Bcrypt is designed specifically for hashing passwords and includes features like automatically generated salt and support for multiple rounds of hashing, which makes it more resistant to brute force attacks than other algorithms.

```
//Hash a password with bcrypt const bcrypt = require('bcrypt'); const password =
```

Security concepts

There are many ways to secure a web application written in Node.js. Some common methods include using secure HTTP headers, using a web application firewall (WAF), and encrypting data.

Secure HTTP Headers:

HTTP headers are name-value pairs that are sent in the request and response between the client and server. There are a number of headers that can be used to increase security, including:

X-Content-Type-Options: This header tells the browser not to try to guess the MIME type of the response, which can prevent certain types of attacks.

Strict-Transport-Security: This header tells the browser to only connect to the server using HTTPS, even if someone tries to connect using HTTP.

X-Frame-Options: This header prevents clickjacking attacks by telling the browser not to display content from this site in an iframe.

Content-Security-Policy: This header allows you to specify which sources of content are allowed on your page, which can help mitigate XSS and other types of injection attacks.

Security concepts

Using a WAF:

A web application firewall (WAF) is a piece of software that filters requests coming into your website or application. It can block requests based on a number of different factors, including IP address, cookies, headers, and more. A WAF can be a good way to add an extra layer of protection to your app.

Encrypting Data:

encryption is a process of transforming readable data into an unreadable format. The data can be transformed back into its original form using a decryption key . When data is encrypted, it helps protect against eavesdropping and tampering . Tampering is when someone modifies data without permission , while eavesdropping is when someone secretly views data .

Security concepts

Adding HTTPS to your Node.js web application is important for two reasons:

1. It ensures that all data passing between your server and clients is encrypted, so that people with malicious intent cannot intercept and read it.
2. It adds an extra layer of security by verifying the identity of your server, so that clients can be sure they are not being redirected to a fake or malicious site.

To add HTTPS to your Node.js web application, you will need to:

1. Get a SSL Certificate. You can either purchase one from a trusted Certificate Authority, or generate a self-signed certificate.
2. Configure your Node.js web server to use the SSL certificate. This will usually involve setting up a virtual host with an SSL port (usually 443).
3. Redirect all HTTP traffic to HTTPS. This can be done using redirect rules in your web server configuration, or by setting up a separate redirector server which sends all HTTP traffic to the HTTPS site.

OWASP (Open Web Application Security Project)-Top 10 Security risks in web applications

1. Injection flaws – accessing and manipulating data entered into web applications through user input, such as via SQL or shell injection. Example code snippet:

```
var username = 'admin'; var password = '1234';
connection.query('SELECT * FROM users WHERE username = ' + username + ' AND pass
  if (err) { throw err; }
  // do something with results
});
```

Example: http://sqlfiddle.com/#!2/fd8be/1

```
'' OR 1=1 LIMIT 1
```

2. Cross-site scripting (XSS) – tricks attackers use to inject malicious scripts into webpages viewed by other users. Example code snippet:

```
<script>alert('You have been hacked!');</script>
```

OWASP (Open Web Application Security Project)-Top 10 Security risks in web applications

3. Broken authentication and session management – weak and easily guessed passwords, session ID vulnerabilities, cookies that are either easily guessable or stolen by third-party attackers.
4. Insufficient logging and monitoring – not tracking application activity or knowing what has happened in the past makes it difficult to determine what is happening on the systems today and makes it more difficult to find and fix issues.
5. Insecure communications – using outdated or unsalted encryption methods, not verifying SSL/TLS certificates, and not verifying message integrity.
6. Broken access controls – granting users too much access, misconfigured role-based access controls, lack of least privilege model.

OWASP Top 10 Security risks in web applications

7. Security misconfiguration – insecure file permissions, revealing sensitive information in error messages, leaving servers and applications publicly exposed without protection.
8. Unvalidated and untested inputs – feeding unvalidated user input directly into web application functions,such as search results, comments, contact forms, etc., can lead to serious security vulnerabilities .
9. Insufficient security controls – failing to deploy standard security measures, such as firewalls, intrusion detection/prevention systems, proper access control measures, etc. can leave an organization’s assets wide open to attacks.
10. Poor software design – insecure coding practices, coding errors that can be exploited for malicious purposes , insecure configuration options ,Race conditions where two threads of execution compete for the same resource.

Authentication

Node.js provides a module called " passport" which helps you authenticate your users easily. You can use different strategies like Local, Facebook, Twitter etc., depending on your requirement. For example,

```
var passport = require('passport'),
    LocalStrategy = require('passport-local').Strategy;
passport.use(new LocalStrategy(function(username, password, done) {
    User.findOne({
        username: username
    }, function(err, user) {
        if (err) {
            return done(err);
        }
        if (!user) {
            return done(null, false);
        }
        if (!user.verifyPassword(password)) {
            return done(null, false);
        }
        return done(null, user);
    });
}));
```

Authorization:

Once the user is authenticated, you need to check if the user has sufficient permissions to access the requested resource. This can be easily achieved using Node's built-in module "acl". For example,

```
var acl = require('acl'); // Define roles
acl.allow([{
    roles: ['admin'],
    allows: [{
        resources: 'blogs',
        permissions: '*'
    }, {
        resources: 'users',
        permissions: ['get', 'put', 'delete']
    }]
}, {
    roles: ['editor'],
    allows: [{
        resources: 'blogs',
        permissions: ['post', 'put']
    }]
}, {
    roles: {
        guest: '*'
    }
}]);
// Check role and permission
```

Python

```
Example 1:
from fastapi import FastAPI, HTTPException
from fastapi.security import OAuth2PasswordRequestForm

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}


@app.post("/login")
def login(form_data: OAuth2PasswordRequestForm = Depends()):

    if form_data.username != "test" or form_data.password != "test":
        raise HTTPException(status_code=400, detail="Incorrect username or passw

    return {"access_token": "fake-super-secret-access-token", "token_type": "bea
```

There are many ways to secure FastAPI endpoints, but some common methods are:

Basic Authentication

```
from fastapi import Depends, FastAPI
from fastapi.security import HTTPBasic, HTTPBasicCredentials

app = FastAPI()

security = HTTPBasic()


@app.get("/users/me")
def read_current_user(credentials: HTTPBasicCredentials = Depends(security)):
    return {"username": credentials.username, "password": credentials.password}
```

Basic Authentication

```
from fastapi import FastAPI
from fastapi.security import HTTPBasic, HTTPBasicCredentials

app = FastAPI()
security = HTTPBasic()


@app.get("/")
def read_root():
    return {"message": "Hello World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, credentials: HTTPBasicCredentials = Depends(security
    if credentials.username == "trudy" and credentials.password == "secret":
        return {"item_id": item_id, "message": "Welcome Trudy!"}
    else:
        return {"message": "Invalid username or password"}
```

Basic Authentication Against Timing Attacks

```python
import secrets
from fastapi import Depends, FastAPI, HTTPException, status
from fastapi.security import HTTPBasic, HTTPBasicCredentials

app = FastAPI()
security = HTTPBasic()

def get_current_username(credentials: HTTPBasicCredentials = Depends(security)):
    correct_username = secrets.compare_digest(credentials.username, "user")
    correct_password = secrets.compare_digest(credentials.password, "pass")
    if not (correct_username and correct_password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect email or password",
            headers={"WWW-Authenticate": "Basic"},
        )
    return credentials.username

@app.get("/users/me")
def read_current_user(username: str = Depends(get_current_username)):
    return {"username": username}
```

ref

https://fastapi.tiangolo.com/advanced/security/http-basic-auth/

Token Based Authentication:

```
from fastapi import FastAPI
from fastapi.security import OAuth2PasswordRequestForm, OAuth2PasswordBearer

app = FastAPI()
security = OAuth2PasswordBearer(tokenUrl="/token", scheme_name="Bearer")


@app.post("/token")
async def get_access_token(form_data: OAuth2PasswordRequestForm = Depends()):
    if form_data.username == "test" and form_data.password == "test":
        access_token_expires = timedelta(minutes=10)
        access_token = create_access_token(
            subject=form_data.username, expires_delta=access_token_expires
        )
        return {"accessToken": access_token}


@app.get("/secure")
@security("Bearer", scopes=["read:users"])
async def read_users():
    return [
        {"id": 1, "name": "Bob"},
        {"id": 2, "name": "Joe"},
    ]
```

OAuth2 and JWT

JWT (JSON Web Tokens)- It is just a token format. JWT tokens are JSON encoded data structures contains information about issuer, subject (claims), expiration time etc. It is signed for tamper proof and authenticity and it can be encrypted to protect the token information using symmetric or asymmetric approach. JWT is simpler than SAML 1.1/2.0 and supported by all devices and it is more powerful than SWT(Simple Web Token). OAuth2 — OAuth2 solve a problem that user wants to access the data using client software like browse based web apps, native mobile apps or desktop apps. OAuth2 is just for authorization, client software can be authorized to access the resources on-behalf of end user using access token.

```
$ pip install pyjwt
```

```
>>> import jwt
>>> encoded_jwt = jwt.encode({"some": "payload"}, "secret", algorithm="HS256")
>>> print(encoded_jwt)
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzb21lIjoicGF5bG9hZCJ9.Joh1R2dYzkRvDkqv3s
>>> jwt.decode(encoded_jwt, "secret", algorithms=["HS256"])
{'some': 'payload'}
```

OAuth2 Authentication (1/7)

```
from datetime import datetime, timedelta

from fastapi import Depends, FastAPI, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext
from pydantic import BaseModel

# to get a string like this run:
# openssl rand -hex 32
SECRET_KEY = "09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30


fake_users_db = {
    "johndoe": {
        "username": "johndoe",
        "full_name": "John Doe",
        "email": "johndoe@example.com",
        "hashed_password": "$2b$12$EixZaYVK1fsbw1ZfbX3OXePaWxn96p36WQoeG6Lruj3vj
        "disabled": False,
    }
}
```

OAuth2 Authentication (2/7)

```
class Token(BaseModel):
    access_token: str
    token_type: str


class TokenData(BaseModel):
    username: str | None = None


class User(BaseModel):
    username: str
    email: str | None = None
    full_name: str | None = None
    disabled: bool | None = None


class UserInDB(User):
    hashed_password: str
```

OAuth2 Authentication (3/7)

```python
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

app = FastAPI()


def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)


def get_password_hash(password):
    return pwd_context.hash(password)


def get_user(db, username: str):
    if username in db:
        user_dict = db[username]
        return UserInDB(**user_dict)


def authenticate_user(fake_db, username: str, password: str):
    user = get_user(fake_db, username)
    if not user:
        return False
    if not verify_password(password, user.hashed_password):
```

OAuth2 Authentication (4/7)

```
def create_access_token(data: dict, expires_delta: timedelta | None = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
```

OAuth2 Authentication (5/7)

```
async def get_current_user(token: str = Depends(oauth2_scheme)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
        token_data = TokenData(username=username)
    except JWTError:
        raise credentials_exception
    user = get_user(fake_users_db, username=token_data.username)
    if user is None:
        raise credentials_exception
    return user
```

OAuth2 Authentication (6/7)

```
async def get_current_active_user(current_user: User = Depends(get_current_user)
    if current_user.disabled:
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user


@app.post("/token", response_model=Token)
async def login_for_access_token(form_data: OAuth2PasswordRequestForm = Depends(
    user = authenticate_user(fake_users_db, form_data.username, form_data.passwo
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username}, expires_delta=access_token_expires
    )
    return {"access_token": access_token, "token_type": "bearer"}
```

OAuth2 Authentication (7/7)

```python
@app.get("/users/me/", response_model=User)
async def read_users_me(current_user: User = Depends(get_current_active_user)):
    return current_user


@app.get("/users/me/items/")
async def read_own_items(current_user: User = Depends(get_current_active_user)):
    return [{"item_id": "Foo", "owner": current_user.username}]
```

ref

https://fastapi.tiangolo.com/tutorial/security/oauth2-jwt/

TBD

1 Introduction to the course, ways to make an online project Medium, linkedin, heroku How to write a readable and informative post 2 Familiarity with latex and project documentation How to write articles in latex, how to write in mathematical language in google colab or in MARKDOWN Overleaf, Markdown 3-5 methods for storing data in large systems (feature analyzes, differences, etc.) Introduction to types of database systems: SQL, NOSQL, KEY-VALUE, GRAPH DATABASE STATEFULL systems REDIS, monogodb, mysql sqlalchemy, pydantic, ORM 6-7 methods for processing data in bulk Working with tabular information, and introduction to Xgoost trees, pandas 8-9 Methods for Integrating Advanced Algorithmics in Pytroch CNN Convolution Network Systems 10 Advanced Testing How to write UNIT TESTS and integration tests Demonstration in Python and BASH 11 Advanced debugging How to debug system, software in multiple processes Methods for debugging in different environments 12 Advanced Profiling What is profiling and how is it profiled in the Linux and Python environment 13 Technological analysis and the connection between business logic and business model in the world of software engineering Business models for software products, the connection between them and the software architecture of large companies (Netflix, Uber, etc.) Overview of the various groups

• Familiarity with modern software development and software methodologies • Meta-programming (ancillary software development tools other than the software code itself) • Debugging and profiling of large systems • Work with tabular as well as NOSQL databases • Introduction to graphical databases • Introduction to advanced frontend and backend systems • Systems design with advanced algorrithmic logic, such as: - Xgboost based solutions - Convolution Networks • Use of fastapi • References to business models for software ventures • Presentation of software projects

Bitcoin mechanics Digital signatures Bitcoin scripts Wallets Consensus protocols (Nakamoto consensus, large scale consensus, VDFs) Ethereum mechanics Solidity Stablecoins and

IPFS

npm install -g ipfs npm install -g ipfs-core npm root -g then dir should look like:

```
 ipfs_demo tree

.
|-- add.mjs
|-- package.json
|-- read.mjs

0 directories, 3 files
```

and the content:

```
(base) ~  ipfs_demo cat package.json
{type:module}
```

IPFS

```
//add.mjs
import * as IPFS from "/home/linuxbrew/.linuxbrew/lib/node_modules/ipfs-core/src
const node = await IPFS.create()

const data = 'Hello World, from EASS part B'

// add your data to IPFS - this can be a string, a Buffer,
// a stream of Buffers, etc
const results = node.add(data)

// we loop over the results because 'add' supports multiple
// additions, but we only added one entry here so we only see
// one log line in the output
console.log(await results)
```

```
//read.mjs
import * as IPFS from "/home/linuxbrew/.linuxbrew/lib/node_modules/ipfs-core/src
const node = await IPFS.create()

const stream = node.cat('AmbRuY12Wk29bV5J9NugxJz2YQeim15siXAg5wR9LFbWd1')
const decoder = new TextDecoder()
let data = ''

for await (const chunk of stream) {
  // chunks of data are returned as a Uint8Array, convert it back to a string
  data += decoder.decode(chunk, { stream: true })
}
```

update the CID in read.mjs

```
node read.mjs
```

Good luck to all of us

EASS
HIT

Be active on EASS discord and try to learn and help each other as much as you can.

Triage

node.js react https://docs.microsoft.com/en-us/visualstudio/docker/tutorials/docker-tutorial https://github.com/docker/awesome-compose/tree/master/fastapi https://luminousmen.com/post/what-are-the-best-engineering-principles https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Concepts gdb profiling debugging ipdb basic security https://owasp.org/www-project-top-ten/ https://owasp.org/Top10/ functional programming https://streamlit.io/gallery + httpx https://github.com/romanvm/python-web-pdb https://codeburst.io/implement-a-production-ready-rest-service-using-fastapi-13f284562c75